Verification and Validation in a Rapid Software Development Process

John R. Callahan and Steve M. Easterbrook
NASA Software IV&V Facility
100 University Drive
Fairmont, WV 26554
304-367-8235
{callahan, steve}@ivv.nasa.gov

Abstract

The high cost of software production is driving development organizations to adopt more automated design and analysis methods such as rapid prototyping, computer-aided software engineering (CASE) tools, and high-level code generators. Even developers of safety-critical software systems have adopted many of these new methods while striving to achieve high levels of quality and reliability. While these new methods may enhance productivity and quality in many cases, we examine some of the risks involved in the use of new methods in safety-critical contexts. We examine a case study involving the use of a CASE tool that automatically generates code from high-level system designs. We show that while high-level testing on the system structure is highly desirable, significant risks exist in the automatically generated code and in re-validating releases of the generated code after subsequent design changes. We identify these risks and suggest process improvements that retain the advantages of rapid, automated development methods within the quality and reliability contexts of safety-critical projects.

1. Introduction

Rapid software development, or rapid application development (RAD), is a broad term characterized by the use of domain-specific computer-aided software engineering (CASE) tools in an iterative process development lifecycle to achieve functional software within short production schedules [1]. First, a basic design is sketched out as a collection of interconnected components using various structured methods . This step defines a basic architecture for a system of interconnected components. Next, the behaviors of some these components and their interactions are defined and implemented. These selected behaviors, often called *features*, are selected on the basis of their priority and utility relative to system requirements. When the selected features have been implemented, the system can be executed (either through simulation mechanisms or code generators) and tested within the scope of the implemented behaviors. Finally, the process repeats itself by enhancing the architecture and implementing the next set of selected features.

Many RAD organizations rely on separate testing group to exercise each partially functional release of the system. In general, these organizations are comprised of two separate but equal subgroups: a “design” group that is responsible for construction of each release and a “test” group that finds problems in each release and works with the design group to fix them. Many errors may remain in each release and it is the task of the test group to find and fix these problems quickly. The test group is responsible for working with the development organization to build revisions to the release.

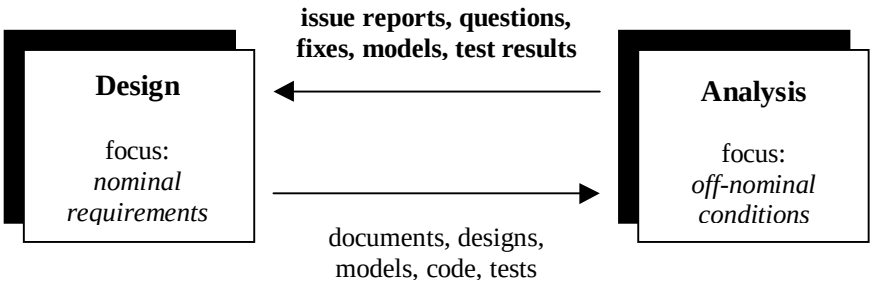

Figure 1: The dialectic between design and analysis groups

The design and test group jointly consider a release to be *stable* if most of the serious problems have been fixed. The process should repeats itself only when a stable release has been achieved. The inability to achieve such stability is an indication of a serious problem (e.g., an incorrect architecture or a poorly designed and implemented feature). Furthermore, the smooth execution of this bipartite process (Figure 1) is essential to the rapid development of software in such organizations. The focus of the design group is typically on nominal behaviors of the system. A nominal behavior is any feature that exhibits an error-free execution of the system. In contrast, the focus of the test group is on off-nominal behaviors of the system. Their analysis should include examination of feature interactions, faults, and unexpected inputs. As organizations attempt to achieve "rapid" development (i.e., deadlines are compressed), this bipartite model becomes important because the focus of the design group tends to become increasingly myopic toward nominal behaviors. The complementary role of the test group usually offsets this tendency and provides a corrective force to the process.

This "build-and-smoke" [2] or “synch-and-stabilize” [3] approach to development is practiced currently in large software development companies because it can deliver functional but incomplete software quickly. Many problems, however, can arise during the process. If the test group does not possess the necessary analysis skills to perform their task, then their contribution is diminished or even dismissed. On the other hand, if the role of the test group is misunderstood by the design group or management does not accept the findings of the test group, then the dialectic between the two groups is pathological. It is management's responsibility to keep the channels of communication open and limited to constructive criticism.

This paper examines a specific case study in which a very large software development organization must interact with an independent verification and validation (IV&V) contractor to achieve incremental, stable releases of software subsystems for the International Space Station (ISS) project. The development contractor plays the role of the design group. They are responsible for the production of stable software releases for ISS subsystems. The IV&V contractor plays the role of test group but applies many types of analysis to the system design and implementation in order to "test" each release.

2. Verification and Validation

IV&V is a *systems engineering* discipline that applies many technical analysis and testing methods to various development artifacts and processes during all phases of the software development lifecycle [4]. Verification is any analysis activity that tries to demonstrate that the product of a phase during development is consistent and complete with respect to the specification before that development phase. Validation is any analysis activity that tries to demonstrate that the product of any development phase is consistent with domain and application

requirements. Both of these activities are important during software development since each phase introduces transformations that may be incorrect with respect to the specifications or the intended utility of the overall system.

A V&V organization can be independent with respect to its technical, financial, and managerial relationships with the design group. A technically independent V&V group uses different tools and techniques than the design group to analyze project artifacts throughout development. A financially independent V&V group may be funded an external quality assurance group or oversight body. A managerially independent V&V group usually reports to the customer or the person above the supervisor of the design group. In general, since independence mostly involves the organizational aspects of analysis, we will frequently use the term "V&V" instead of "IV&V" to describe the analysis activity itself.

V&V analysts also play an important role during maintenance because a significant portion of maintenance tasks involve functional enhancements to the behavior, design, and implementation of a system [5]. V&V's primary task is to manage project risk by identifying and monitoring errors throughout the development and maintenance process. Since it is impossible to identify and resolve all errors early in a project's lifecycle, it is the V&V contractor's task to identify errors as early as possible and track the progress towards their resolution. For example, a minor design error may be ignored early in the process if the developer believes that a yet-to-be-designed feature will solve the problem.

During development and maintenance, IV&V maintains a list of reports on problems found during the development process [6, 7]. It tries to verify solutions to these problems and produces reports on new problems when necessary. Such reports come in a wide variety of formats an include items such as change requests (CRs), discrepancy reports (DRs), problem reports (PRs), issue reports (IRs), and issue tracking requests (ITRs). Most of these reports are authored by an IV&V contractor during development but can originate in the design group as well. Each report has a disposition that changes as a problem is addressed throughout development. V&V will track problem reports and ensure that each report is eventually addressed at an appropriate point during development. If the problem is not adequately addressed, then V&V can report this up the management hierarchy. In most cases, however, this route is avoided. Most problem reports remain as part of the normal dialogue in confidence between the design and V&V groups.

3. Case Study

Exploration of space requires the use of sophisticated software with high levels of quality and reliability. On the International Space Station (ISS) project, one contractor decided to use the MatrixX[1] tool to reduce development costs and improve design quality. The tool was used to develop and perform white-box testing (unit and integrated component level verification) of human-rated critical flight software. The tool is extremely useful in designing and generating code for complex systems. Our task was to identify process issues related to the tool's use in the ISS effort and suggest paths for achieving the highest possible quality and reliability via testing during the development process.

Figure 2 is a high-level conceptual model of the production process for each software release. The ISS software design is comprised of several computer software configuration items (CSCIs) onto which are loaded one or more computer software components (CSCs). Each computer software component is designed as a hierarchically nested set of computer software units (CSUs)

[1] MatrixX is a trademark of Integrated Systems Inc.

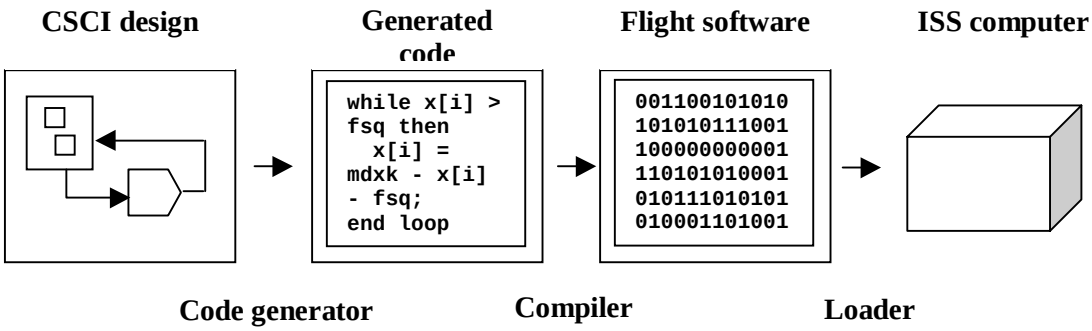

Figure 2: Release production process

that can be tested in isolation through simulation in the design tool. Source code (in Ada) can be generated automatically for an entire CSCI using a code generation tool. The generated code is then compiled to the platform using a conventional compiler to produce the executable flight software and loaded onto a flight computer.

The contractor employs the design tool in a rapid software development process that calls for iterative design and testing of the high-level design and releases of the automatically generated code for each CSCI. Testing was planned for only the high-level CSCI design through simulation (a capability of the design tool) and the code generated for the entire system, but not of the automatically generated code at unit level. This was felt to be too expensive and redundant. Critical CSUs would also be tested at the unit level through simulation in the design tool. While the ability to simulate the design at the CSCI and CSU levels is extremely useful and can lead to the discovery of early design flaws, some errors can only be detected through analysis of the generated code:

- The code generator is not a certified, verified tool. As with any new tool, documented flaws have been shown to exist in the code generator. These flaws lead to errors in the translation process and could introduce problems in the source code that do not exist in the high level design. While visual code audits of the generated code may be useful, they are expensive and poor substitutes for unit testing.
- Important objectives of unit testing are to ensure that adequate test coverage is achieved and that extreme and singular values for variables are tested. Unit testing of the high level design cannot achieve this in generated code that may introduce auxiliary and temporary tasks, procedures,, and variables.
- For any number of reasons (data storage, real-time processing, etc.), the design environment may not behave as the actual environment will under identical test inputs. It cannot be assumed that they will behave similarly in all situations.

The developer argued that while these concerns were valid, they did not represent significant risk to the project when balanced against the productivity and design quality gains. The V&V contractor, however, argued that these were significant risks that must be addressed in the short and long term. It was our task, as a research team, to help identify the issues while seeking to preserve use of the code generator without undue impact on project costs, schedule, and functionality.

3.1 Problems

The use of code generators to produce flight software creates several testing and maintenance problems. The problem originally focused on unit testing aspects of the generated code, but quickly expanded to other areas of concern:

- The sequencing characteristics of the code generator creates problems related to long-term maintenance of the software. When changes to the software design are necessary, they will be done in the high level specification. Since the code generator uses data flow analysis to produce a single, interleaved module for an entire system, small design changes are highly likely to produce significant changes in the entire body of generated code. As a result, a significant of amount of regression testing of the generated system will be required for even small changes. The cost of this additional testing threatens to erode cost savings realized by use of the tool in the first place.
- The code generator makes any errors found during testing and mission operations difficult to isolate and debug. Errors caused by problems other than design flaws cannot be debugged in the high level design. Problems other than design flaws may require changes to the generated code itself (e.g., such cases did occur in practice on this project). Such changes create a divergence from the design and enormous configuration control problems in the short and long term.
- The automatically generated code does not comply with Ada coding standards. While the tool vendor never intended for generated code to be read or altered, *such changes are occasionally necessary to access functions not available in the high-level design environment*. Furthermore, readability of the generated code is desirable because the system much often be debugged at this level. Thus, some form of structuring and coding standards are necessary to ensure readability, maintainability, reliability, reusability, and portability. It is highly unlikely that over the entire lifecycle of the system, the code will remain unexamined. Based on the size of the overall project and experience with other large flight software projects, it will become necessary at some point to examine the generated code.
- The code generator currently produces inefficient code in terms of size and memory usage. While the manufacturer is improving the technology of the tool (a problem itself - see below), it was estimated that the ISS GN&C will generate 12,382 SLOCs requiring 386K for storage. The current design produces over 120,000 SLOCs and would have required secondary storage in addition to the original 1MB EEPROM for each MDM. Before a recent redesign initiated by the identification of this problem, this code bloat may have dictated a significant hardware change that would have significantly increased system fault risks (i.e., seek delays and potential failures of the secondary storage unit).
- There existed no satisfactory configuration control plan for long-term evolution of the design specification, generated code, testing, and new releases of the code generator tools. Plans and processes need to be developed with regard to upgrading to new tool releases, how this affects code generation, unit and system level testing.

These problems are likely to occur with the use of any high level design and code generation tools including modern programming language compilers, linkers, and loaders. It was our task to help identify these issues and develop approaches to mitigate risk while leveraging the advantages of the tools.

3.2 Solutions

Based on these findings, the development and V&V contractors worked together to develop a comprehensive plan that leverages the productivity gains of the tool while integrating quality and reliability goals. Some of the adopted plans include:

- Heavy use of generated code modularization. The code generator contains features for isolating the generation of code for specific design units into modules. This feature introduces code bloat, but significantly reduces coupling in the generated code.
- Unit testing of generated code. It was decided that the modularization features made it possible to effectively unit test the generated code. This includes assessment of code coverage adequacy based on path/logic coverage, data minimum/maximum values, and erroneous data inputs.
- Auto-generation of test cases. One additional benefit of code modularization was that tests on the design specification could now be used to generate system-level test cases.
- Adoption of a configuration control, test, upgrade and integration plan. This plan includes a modest upgrade path combined with regression tests for modules that are expected to be affected by changes to the code generator and design changes. The reduction in coupling means that newly integrated design modules no longer have ripple effect on the rest of the generated code and can be tested in isolation from the rest of the system specification.

Our recommendations focused on the use of design modularization features to achieve and maintain fidelity between the design and generated code. Although not easy to use and not enforced at the design level, the modularization conventions have been extremely useful in facilitating the iterative design and analysis process. Indeed, the tool manufacturer is planning to incorporate enforcement of modularization in subsequent releases of the tool.

4. Summary

Automated tools will continue to be used with increasing frequency in software development projects for many good reasons including cost, quality, and productivity. Indeed, our analysis supports the continued use of CASE tools, but we must be continuously aware of the risks associated with new technologies that are co-evolving with our projects. We found that it is possible to integrate safety-critical goals of quality and reliability during the development process if existing organizations include complementary advocates for nominal behaviors (the designers who want to see the software achieve specified functionality) and advocates for off-nominal behaviors (the V&V team who want to ensure that rare cases have been accounted for as best as possible). If both teams work together within an iterative process that facilitates both design and analysis, then concomitant goals can be achieved.

Many of our suggested improvements were based on the need to reduce process risks as well as reducing the risk of errors in the product. The benefits of V&V analysis can only be leveraged if the turn-around time for analysis can be streamlined. The modularization and configuration control plans greatly enable V&V to provide timely and useful analysis to the design group. Our recommendations helped reduce the tremendous amount of rework that would have been necessary to maintain a productive dialogue between the two groups.

Finally, it is V&V's continual task to monitor the co-evolution of CASE tools used on the project by analyzing the differences in generated code between tool versions. Analysis of these differences will provide useful information for tailoring the verification process to accommodate known differences between the design and deployment environments. Depending on changes to the tool, extensive regression tests may be necessary for some project releases because of the impact on the generated code regardless of modularization boundaries.

5. Acknowledgements

We would like to thank Tom Marhsall, Jim Dabney, and Dan McCaugherty of Intermetrics, Inc. for their help in collecting and summarizing this data. We are most grateful for their time, hard work, and dedication to quality performance of V&V tasks. This report has been prepared under NASA Cooperative Agreement NAG2-797 through funding from NASA's Office of Safety and Mission Assurance and the NASA/Ames Research Center.

6. References

1. McConnell, S., *Rapid Development: Taming Wild Software Schedules*. 1996, Redmond, WA: Microsoft Press.
2. McConnell, *Daily Smoke and Build Test.* IEEE Software, 1996. **13**(4): p. 144.
3. Cusumano, M. and R. Selby, *Microsoft Secrets*. 1995: The Free Press. 512.
4. Lewis, R., *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*. 1992, New York: John Wiley & Sons. 356.
5. Garlan, D., G.E. Kaiser, and D. Notkin, *Using Tool Abstraction to Compose Systems.* j-COMPUTER, 1992. **25**(6): p. 30-38.
6. Callahan, J. and G. Sabolish. *A Process Improvement Model for Software Verification and Validation.* in *The 19th Software Engineering Workshop*. 1994. NASA Goddard Space Flight Center, Greenbelt, Md.
7. Callahan, J., T. Zhou, and R. Wood. *Software Risk Management through Independent Verification and Validation.* in *4th International Conference on Software Quality (ICSQ 94)*. 1994. McLean, Va.